# Real theorem provers deserve real user-interfaces

Laurent Théry, Yves Bertot, Gilles Kahn

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Sophia Antipolis
B.P. 109
06561 Valbonne Cedex
France
Tél.: 93 65 77 77

Rapports de Recherche

N°1684

*Programme 2*
*Calcul symbolique, Programmation*
*et Génie logiciel*

# REAL THEOREM PROVERS DESERVE REAL USER-INTERFACES

Laurent THERY
Yves BERTOT
Gilles KAHN

Mai 1992

# Real Theorem Provers Deserve Real User-Interfaces

Laurent Théry — *University of Cambridge*
Yves Bertot, Gilles Kahn — *INRIA – Sophia Antipolis*

**Abstract** –This paper explains how to add a modern user interface to existing theorem provers, using principles and tools designed for programming environments.

# Une stratégie de construction d'interfaces utilisateur pour les systèmes de preuves assistées par ordinateur

**Résumé** – Cet article propose une méthode pour ajouter une interface utilisateur moderne à un système de preuves assistées par ordinateur préexistant. Cette méthode est fondée sur des principes et des outils originellement conçus pour les environnements de programmation.

# Real Theorem Provers Deserve Real User-Interfaces*

Laurent Théry
*University of Cambridge*

Yves Bertot Gilles Kahn
*INRIA – Sophia Antipolis*

**Abstract**

This paper explains how to add a modern user interface to existing theorem provers, using principles and tools designed for programming environments.

## 1 Introduction

There are a number of reasons for which it is interesting to build better user interfaces for theorem proving systems, i.e. computer systems that assist in making formal deductions.

- First the user-interface of today's theorem proving systems is generally very weak. This is in contrast with many other facets that have matured considerably in the last ten years. Notable user-interface experiments are (IPE[Ritchie88], Mural[Jones91]) but they put forward simultaneously a new logical system *and* a new interface.

- Proving theorems is an exemplary symbolic activity. It is generally accepted that a co-operation between the user and the computer is mandatory, even though some subtasks may be handled entirely automatically. In comparison to symbolic algebra systems, such as Mathematica[Math] or Maple[Maple], theorem proving systems add an interesting ingredient of proof planning and management. But many ideas in this paper apply directly to the building of user interfaces for symbolic algebra systems, in the spirit of Kajler[Kaj92].

- Producing proofs may become a routine part of a software developer's task. Today, emphasis on proofs comes from several specialized disciplines such as circuit design, protocol design, and time critical software. But in the future, in areas where formal specifications are feasible, mechanical proofs will be the natural reward. For this to happen though, it is necessary to invest in the engineering aspects of proof construction: software developers are not expected to become theoreticians. They are expected to focus their efforts on design and to perform proofs along the way as the natural way of making their ideas precise.

- Building a program that meets a certain specification is, in a very strong sense, similar to providing a constructive proof of that specification. Hence it is hoped that ideas about the engineering of proof construction will bring new light on program development.

*This work was supported in part by the "Logical Frameworks" Esprit Basic Research Action. Laurent Théry is supported at Cambridge by SERC grant GR/G 33837 and a grant from DSTO Australia.

## 1.1 Basic premise

Building a new theorem proving system is an ambitious task that must be founded either on new theoretical insights or on new system ideas. It is unreasonable to do so merely to experiment with user interfaces. There are already a number of theorem proving assistants, and most of them have an organized community of users. Thus, the basic premise of our work is that *we shall not build a new prover.*

Theorem provers are implemented in a variety of programming languages (e.g., dialects of Lisp or ML, $\lambda$-Prolog, etc.). Linking a sophisticated user-interface with large programs written in these languages is likely to be technically difficult and unpleasant. Furthermore, the resulting system will be bigger and, most likely, far less portable. An alternative solution is to build the user interface as a separate entity, comprising of one or several processes, communicating with the theorem prover via a *protocol.* There are clear advantages: the user interface can be implemented in any appropriate language, it may run on a separate machine, it will induce one to isolate generic components of interest for any theorem prover. But this solution also raises issues, addressed in this paper: is there an efficiency penalty in this separation, is it necessary to develop more code on its behalf, does the prover allow an implementation of the desired protocol?

The analysis presented in this paper is based on experiments pursued with three fairly different, publicly available systems: Felty's prover[Felty89], implemented in $\lambda$-Prolog, HOL[HOL88], implemented in ML and Lisp, and Isabelle[Isa90], implemented in SML. All three of them have a programmable top-level, so that it is possible to implement the prover's side of the protocol while remaining a conventional system's user.

The experimental user interfaces were created using the Centaur system [Centaur92] as a toolkit. While the core of the Centaur system is implemented in Lisp, it is a meta-system and most of the needed code was written in one of Centaur's meta-languages.

## 1.2 An example: the HOL front-end

To give a concrete example of the kind of interactive systems we envision, we begin with a very brief description and a sample session.

### Overview

The system includes two separate entities: the user-interface and the proof engine. The user, in principle, never communicates directly with the proof engine and, conversely, all proof engine output is presented by the interface.

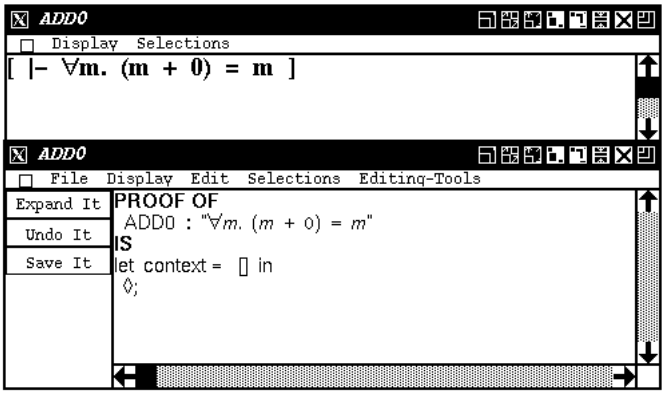
The interface includes several windows displaying, typically, the proof under development, the remaining goals (with special emphasis on the goal currently under attack) and a variety of menus, that may be of help in constructing the next step of the proof. To compose this step, the user may use the keyboard and the mouse in a fairly free combination. When the next proof step is ready, it is recorded in the proof script and shipped to the proof engine. In turn, the proof engine replies with a number of messages that result in updating the interface's windows. When the proof of a fact is complete, it is recorded in the proof engine's database. This fact is available as a theorem for further work during this or later sessions.

### A session

Here is a complete proof session for the trivial theorem `ADD0`:

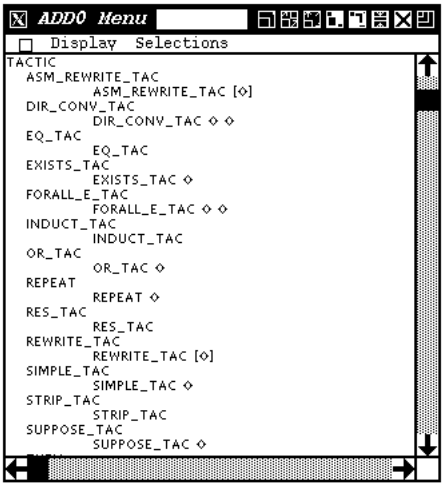
$$\forall m\ m + 0 = m$$

using our current user-interface for HOL. In this interface, the user may decide to start a proof at any time. To begin a new proof, the user clicks on a button "New Proof", which pops up a dialog window requesting the statement of a theorem and a name for it. Then two new windows appear on the screen:



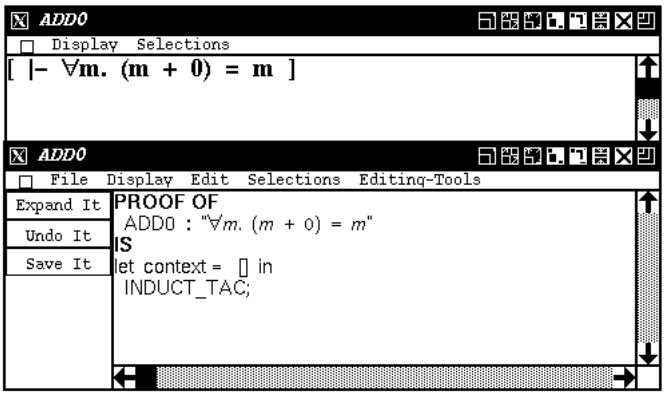

The top window contains the facts that remain to be proved. It is called the *subgoal* window. Initially, the subgoal window contains the statement to be proved. The bottom window contains the text of the proof so far. It is called the *script* window. Commands to the prover are inserted in this window. They are called *tactics*. Initially, the command is just a place holder[1] ◇ .

The user constructs a new command by editing an arbitrary place holder in the proof script. When the command is ready, selecting the button "Expand it" performs it, or *expands* it to use the HOL terminology. To help in constructing the command, a menu displays available tactics and their argument structure:

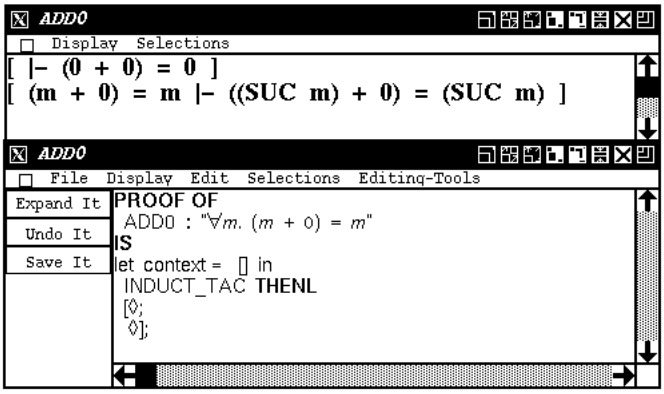



Here, the goal should be proved by induction on $m$, thus we select **INDUCT_TAC** in the menu. The script window is automatically updated:

[1] For the use of the constant `context`, which is immaterial in this example, see in section 4 the discussion of occurrences.



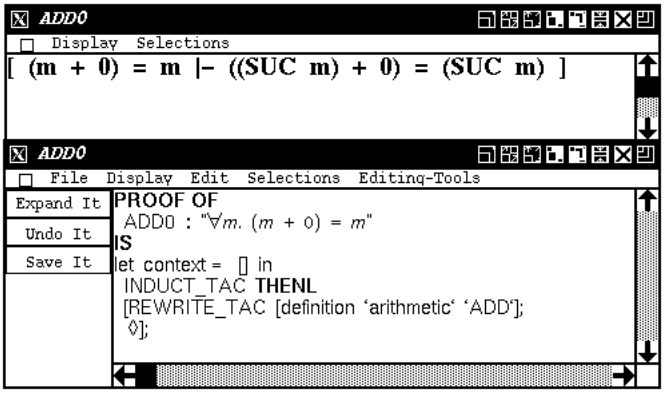

Depressing the "Expand it" button requests execution of the command. Both the subgoal window and the script window change. The subgoal window displays two new subgoals, corresponding to the base case and the inductive case of the proof. In the script window, two place holders are introduced, to be filled with the proofs for the two subgoals:



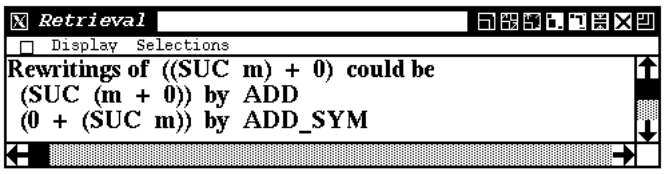

To prove the goal $0 + 0 = 0$, the user edits the first place holder. To prove the second goal, $m + 0 = m \vdash (SUC\, m) + 0 = (SUC\, m)$, the second place holder must be edited. Suppose we want to prove the base case first. This trivial fact follows directly from the definition of addition. We replace the first place holder with the expression

```
REWRITE_TAC[definition `arithmetic` `ADD`]
```

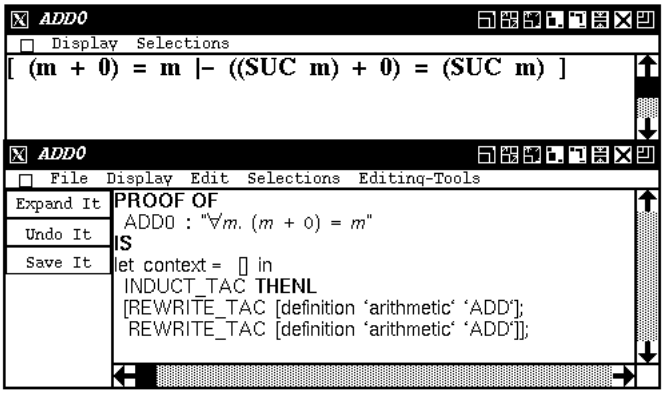
and after expansion we have only one subgoal left in the subgoal window, and the corresponding place holder in the script window



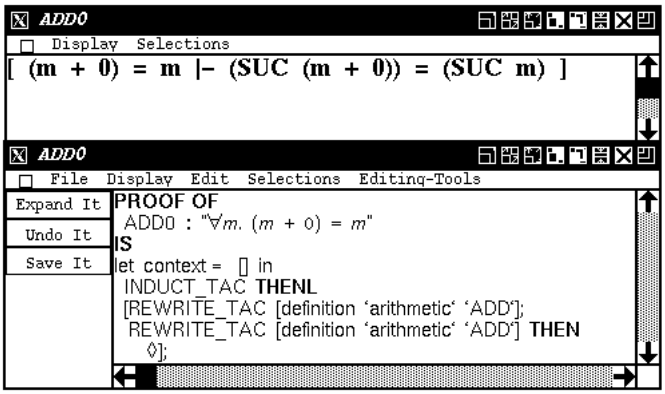
To prove the inductive case, we could use the same tactic. Instead, let us use a *retrieval* tool. Given a term, this tool displays its possible rewritings with respect to the theorems and definitions present in the database of the theorem prover. Here, we select $(SUC\ m)+0$ in the subgoal window, obtaining in the retrieval window the following answers:




Now selecting the first answer in the retrieval window provides the full name of the definition, which is needed in the script window

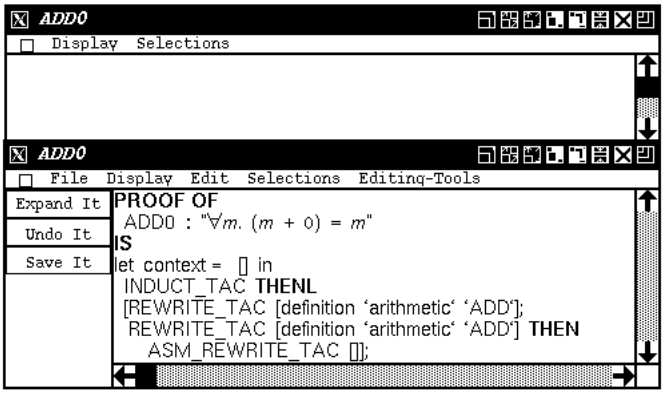



Now expanding this proof step is not sufficient to prove the goal, so a new place holder appears in the script window:




Here the conclusion is a trivial consequence of the assumption. The tactic ASM_REWRITE_TAC[], which rewrites the conclusion using only the assumptions is sufficient.

There are no remaining subgoals in the subgoal window nor any place holders left in the script window. So the theorem has been proved, and the script window contains a complete tactic that proves the theorem.




In the rest of this paper, we discuss the technical problems encountered in constructing the kind of user-interface demonstrated above.

# 2 Generic Interfaces for Symbolic Systems

This section describes how information gets from the user to the theorem prover and back, across the user interface. At this stage, exactly which theorem prover is used is of no consequence. In fact, the communication could be established with any kind of symbolic computation system. We first discuss communicating individual fragments of structured data. Then we consider the complete dialog between the user interface and the symbolic engine. Last, we show how to build a parallel and easily extensible interface.

## 2.1 Data interchange

A symbolic engine expects commands and returns results. It has a fairly crude interface that is not appropriate as the basis for a communication protocol.

### From symbolic engine to interface

To begin with, the results are printed in a manner intended for the human reader. To make such results *available* in the interface —where we want to do more than display them— they must be parsed. This can be complex when the symbolic engine uses arbitrary syntactic conventions including the use of subscripts and superscripts.

In fact, however, it is more practical to alter the symbolic engine's output function, so that it prints expressions that are easy to parse, but would be unacceptably verbose for a human reader. As an example, here is the formula produced by the modified HOL for $\forall m, n\ m + n = n + m$:

```
(COMB (CONST !)
      (ABS (VAR m (num))
           (COMB (CONST !)
                 (ABS (VAR n (num))
                      (COMB (COMB (CONST =)
                                  (COMB (COMB (CONST +) (VAR m (num)))
                                        (VAR n (num))))
                            (COMB (COMB (CONST +) (VAR n (num)))
                                  (VAR m (num))))))))
```

Of course, this implies modifying the symbolic engine's top level loop, which is usually possible. Then, all we need is to build a pretty-printer in the user-interface. This task is fairly easy; in particular, Centaur has a meta-language called PPML for describing pretty-printing rules.

#### From interface to symbolic engine

It is also necessary to send data from the user-interface to the symbolic engine, for example when using as argument, in a command, a fragment of a formula that was produced earlier. It is *a priori* simple to format this data in a way that will be acceptable to the symbolic engine's input parser. But the information normally present in the symbolic engine's terse output may not be sufficient to reparse; extra type information may be needed. Thus, the symbolic engine must send some type information along with a formula, that is not used for display by the user-interface, but only for later communication back to the symbolic engine. On the other hand, too much type information will slow down communication. The example above shows that, in communicating with HOL, it is sufficient to include types for variables only.

In fact, these remarks indicate that the designers of symbolic engines should specify an abstract syntax for the objects they manipulate, together with a standard textual representation of terms of the abstract syntax. Clearly, this would not be of interest solely for building user interfaces.

#### Typing in data

The user also needs to enter expressions as text. If it is easy to reproduce the symbolic engine's parser, there is no difficulty. But in fact, this is *not* necessary. We can use the symbolic engine's parser. The text to parse is simply sent to the symbolic engine within a command, which resends the same data using the standard protocol.

### 2.2 Multiplexing and de-multiplexing

The output of the symbolic engine is made of information of different kinds, that the user-interface will have to dispatch to the proper window. For example, there may be error messages, goals, theorem statements, etc. We say that the engine's output channel is multiplexed. Here again, it is unreasonable to have the user-interface parse its input to guess which is which. The top level loop of the symbolic engine is best modified to surround each kind of data with unambiguous text markers. It is then easy to scan the output stream of the symbolic engine and trigger the actions corresponding to each message, with the message's text as argument. In fact, we handle this in an even more uniform fashion. The protocol is specified by a list of quadruples: begin marker, end marker, parser to use for the enclosed text, kind of message to generate for the interface's use. The de-multiplexing package sections its input using the markers, parses the data using the corresponding parser, and stores the result in a message of the appropriate kind, which is broadcast to the interface.

For reasons of simplicity and efficiency, this protocol is absolutely *stateless*. In our experiments so far, this has been sufficient. It is easy to see how we could take into account a more sophisticated protocol, since certain parts of the user interface are already built in Esterel[CI89]. In particular, the current protocol is completely *asynchronous*. In most cases, this is adequate. But difficulties come up, for example, when one wants to provide a means of interrupting the symbolic engine. Clearly, this cannot be implemented in a purely asynchronous manner. One technique is to use a different mode of communication, sending for example a Unix signal to the symbolic engine and

putting a special *marker* on its input stream. The signal forces the symbolic engine to stop whatever computation it is involved in; the signal handler then flushes all commands in its input up to the marker, and sends a message of acknowledgement to the front end that allows traffic to resume.

Such a protocol involves more programming on the symbolic engine's side than we would care to develop. This is due in part to the lack of precise specification of what is actually going on when a conventional user interrupts the symbolic engine.

### 2.3 Architecture of the user interface

The architecture of the user-interface itself is not different from that of any software development environment. We follow the principles stated in [Clement90] and [CMP91], organizing the interface as a collection of independent components, plugged on buses and communicating via messages. One component is in charge of all communication with the symbolic engine. In particular, it demultiplexes the symbolic engine's output, sending messages on the appropriate buses. This architecture is well adapted because:

- it is very flexible. As we experiment with ideas in user-interface, it must be very simple to add a new component, for example a new kind of menu.
- it allows dynamic evolution. For example, when the user starts a new proof, which can happen at any time during a session, new interactors are created and added to the existing networks.

Several parts of the user interface are really not specific to a particular symbolic engine nor, in fact, of this kind of application. Consider for instance a theorem prover that includes a *theory library* that permits one to group theorems into packages, in the way a program library would organize package specifications. The part of the interface that displays existing theories upon request and maintains consistent the state of these theories, as they are updated by the user, is a substantial reusable component.

Note that even though the user-interface may be implemented in a strictly sequential system, the network architecture gives *de facto* a feeling of parallelism. The user can work on two tasks separately, but the messages between objects corresponding to these two tasks will actually be interleaved. Moreover, it is possible to connect the user-interface to several symbolic engines simply by having one connection component for each.

## 3 Building a proof

We proceed now to discuss technical problems that are more specific to theorem proving environments: goal directed search and progressive construction of a proof script. But this is not so arcane: an environment for constructing programs meeting formal specifications would probably raise similar issues.

### 3.1 Displaying the goal(s)

Displaying goals that remain to be proved is a central task of the user interface. Goals are algebraic-logical formulae with a hierarchical structure, just as programs are. Hence we can take advantage of the pretty-printing facilities provided in the Centaur system. However, some distinctive traits of symbolic computation push the generic pretty-printing technology to its limits in three areas: typesetting, incremental handling of large subgoals, and extensibility of the interface.

### Typesetting

The typographical style used in any formal setting is rich: it makes use of special symbols, such as the quantifiers $\forall$ and $\exists$, the implication sign $\supset$, but also indices and exponents, and, routinely, two-dimensional arrangements such as the formula $\sqrt[3]{a^{\sqrt{b}}}$. These characteristics require a powerful two-dimensional display software, using variable width characters. In contrast to TEX, it is mandatory to handle automatically formulae that are very long. Of course, selecting a sub-expression with a single mouse click, as well as dragging the mouse over a list of expressions must be provided.

The Centaur system [Centaur92] provides two display engines. The standard package is adapted to program text and simple formulae; the `Figue` package handles two-dimensional layout. In both cases, the layout of a formula is computed from a PPML specification, which is made of rules of the form

$$\text{pattern} \rightarrow \text{format}$$

such as the following one

```
plus(*x,*y) → [<hv 1,tab,0> *x in class=binop:  "+" *y]
```

The left hand side of the rule contains a tree pattern, where variables are prefixed by the character `*`. The right hand side specifies layout and may be paraphrased as follows: *Display the tree* `*x`*, followed by the string* `+` *in class* `binop`*, followed again by the tree* `*y`*. Separation between these elements should be 1 unit of white space if it all fits on the line; if necessary, go to the next line, indenting by* `tab` *units of space.*

The rules also specify precisely the mouse selection mechanism. Here is roughly how mouse selection works. From the position of the mouse, a token is identified, then the rule which immediately produced that token, then the tree occurrence that matched the left hand side of that rule.

The use of symbolic classes allows one to associate colour and font information to tokens such as the string `+`. This association is provided by the final user in a *resource file*, in the familiar X-window system style. Examples of output, in black and white, can be seen in Fig. 2.

### Incremental handling of large subgoals

During the course of a proof, subgoals may grow to be quite large, i.e., they may fill several screenfulls. The user wants to scroll over the text of the goal with scrollbars and use elision to shrink subexpressions. Here, traditional ideas from structure editors, such as imposing a general or a local level of detail on a formula, give the user adequate control: it is possible to reduce an entire sub-formula (or a sublist within a formula) to an abbreviation symbol with a simple mouse gesture.

Large subgoals create a performance problem. First the proof engine must send a large formula to the front-end, which generates a large flow of data in the network. Further, upon receiving this large amount of data, the user-interface must compute its representation and display it. As data transfer and display must take place in that order, the whole interface slows down.

To solve this problem, we notice that, except for the initial goal, every subgoal is created by its predecessor and usually shares a non-negligible common part with it. For example, most assumptions are shared from one subgoal to the next. Hence we can use two techniques:

1. When displaying the new subgoal, the user-interface, which has retained the previous subgoal, can take advantage of incremental display to minimize re-computation and redisplay,

2. When creating a new subgoal, the proof engine can send only the difference with the previous subgoal.

In Centaur the first technique is automatically available because both display engines are incremental. There are two sources of incrementality: first, the computation of the desired layout, which PPML formalizes as a *regular tree translation* from the formula to a tree of geometric boxes is incremental, yielding improved performance. Second, the mapping of this modified geometric structure to the screen is also incremental. This results in a much better stability of the screen, an absolutely essential property for an interface. Notice that this technique is available even if the proof engine sends complete goals: the user-interface simply computes the difference between the formula it is supposed to display and what is currently displayed, then performs the incremental redisplay.

The second technique implies an extension of the communication protocol between the proof engine and the user-interface. This extension must be implemented, in part, on the proof engine's side, because we want to minimize traffic.

For this protocol, there are two possibilities. The first one is to emit a modification message that describes how the new goal can be built from its predecessor by transformation. So the communication protocol for subgoals must contain instructions such as `delete`, `change`, `add`. An alternative is to use the previous goal as a reference formula, and to give a functional description of the new goal using references to its predecessor whenever possible.

As an example of the difference between these two protocols, assume that the current goal is:

$$A, B, C \vdash D \Rightarrow E \tag{1}$$

where $\vdash$ separates assumptions from the conclusion. Here $A$, $B$, $C$, $D$, and $E$ stand for possibly very large expressions. All systems allow you to add $D$ to the assumptions and make the natural transition to:

$$A, B, C, D \vdash E \tag{2}$$

In the standard protocol, where complete goals are communicated to the interface, the message describing subgoal (2) would be:

```
(goal (conclusion E) (assumptions A B C D))
```

The first protocol describes how the contents of the goal change:

```
((1.change[E]) (2.add[D]))
```

Using the second protocol, the proof engine emits:

```
(goal (conclusion (ref 1.2)) (assumptions (ref 2.1) (ref 2.2) (ref 2.3) (ref 1.1)))
```

where `ref` takes as argument a path in the previous subgoal.

Both protocols reduce drastically the traffic due to large subgoals. This is *crucial* to using our distributed approach for real proofs.

The first protocol maps quite directly to the incremental display engines. On the other hand, the second protocol is more homogeneous and somewhat easier to implement on the proof engine's side.

### Extensibility

Centaur provides tools to manipulate and display abstract syntax trees. When defining the formal language to represent logical formulae, we must decide how much information should be wired in the abstract syntax of the language.

For example, suppose the language contains a `+` operation, such that if the strings $E_1$ and $E_2$ represent two well-formed expressions $e_1$ and $e_2$, then $E_1$+$E_2$ also represents a well-formed expression $e$. We have to choose between considering `+` as a regular binary operator of the language, constructing the tree $e = plus(e_1, e_2)$, or considering the expression as the mere application of a function to one argument, this function being itself obtained by applying a function to another argument, thus constructing the tree $e = @(@(+, e_1), e_2)$ where @ denotes function application.

The first choice gives simpler trees. We say that the `+` operator has been *internalized*. However, this has the drawback that the language is hard to extend. Adding a new operator (say `|` to express the divisibility relation) means changing the set of operators in the language, a main change in the data type structure of the system. In the context of theorem proving, where developing new notation is critical [Griffin88], this solution is inadequate.

The second choice gives more complex trees, but the abstract syntax is fixed once and for all with a minimal set of operators. New operations like `|` can simply be referred to by an identifier. In our interface for the Isabelle theorem prover, the abstract syntax contains the following operators: `@` for application, `l` for lambda-abstraction, operators to represent variables and literal values, a list-forming operator, and an operator to represent inference rules. Infix operators are represented as function symbols.

The display mechanism uses resources provided by the user to distinguish infix operators from regular functions, to decide the precedence of operators, and to choose symbolic representations. For example, the tree

$$@(\texttt{All}, \texttt{l}(m, @(@(\texttt{eq}, @(@(-, @(@(*, m), \texttt{one})), m), \texttt{zero}))))$$

which would otherwise be represented as

$$\texttt{All}(\lambda m.\texttt{eq}(-(*(m, \texttt{one}), m), \texttt{zero}))$$

is actually displayed as

$$\forall m.\, m \times 1 - m = 0.$$

In the user's resources, `All` appears as a binder, with symbolic representation $\forall$. Function symbols `*` and `-` appear as infix operators with symbolic representations $\times$ and $-$. The precedence given in the resources indicate that no parentheses are needed here. Literals `one` and `zero` have symbolic representation 1 and 0.

Note that two additional issues must be weighed carefully when designing the abstract syntax: what subexpression should the user expect to see selected, when the mouse points at a given symbol; how does one translate a path in the interface's abstract syntax tree to an occurrence for the theorem prover (cf. section 4.2).

## 3.2 Controlling the proof construction

We now examine the abstractions involved in steering the proof. A proof starts with an initial goal. Then to prove this goal, one applies a tactic that breaks it into several, presumably simpler, subgoals. During proof construction, a user may select:

1. which remaining subgoal to attack
2. what tactic to use for this subgoal.

Most of the standard interfaces only allow the user to choose a tactic. The choice of subgoal is automatically performed by the prover, usually by a simple method (such as taking the first element in the list of pending subgoals). It is preferable to avoid rigid strategies and give more freedom to the user. In a prover with logical variables (such as Felty's or Isabelle) this possibility is essential to obtain simpler or/and more natural proofs: as a logical variable may occur in several subgoals, it is usually better to attack subgoals that have a chance to instantiate it before attacking other subgoals.

It is easier to give such freedom when the prover maintains an explicit *proof object* that represents the current state of the proof. Then one provides simultaneously a tactic and a path in the proof object that locates the subgoal to which we want the tactic to be applied. This technique is used for example in HOL, thanks to a special subgoal package that maintains a proof object.

In a prover where there is no explicit proof object, alternative solutions may still be designed. For example, in Felty's theorem prover, the state of the proof is only represented by the list of remaining subgoals. But associated to each subgoal in this list is the path that it would have if it were embedded in some proof structure. This path represents the minimal information concerning the proof structure that must be maintained to have an acceptable subgoal selection mechanism. In a sense, this path is a symbolic name given to a subgoal, and it remains unchanged as the proof unfolds in other directions.

As soon as we want to control manually the construction of a proof, we must provide the user with a way of *undoing* proof steps. There are at least two ways of understanding what the user may want. The first way is purely chronological and is usually present in modern interactive systems. Undoing is simply understood as restoring the system to the immediately preceding state. The second way is more related to the fact that proof construction is a kind of parallel process. As we may interleave proof steps relating to different subgoals, it makes sense to undo with respect to a given subgoal, i.e., to undo the last step that led to the existence of this subgoal, even though this may not be the last step in the sequence of actions of the user. We call this operation *local undo*.

If historical undoing is quite simple to implement in a traditional prover, local undoing is more difficult, in particular if the prover deals with logical variables. Without logical variables, proofs of different subgoals can be considered as truly parallel activities that do not interfere with one another. As soon as there are logical variables, a proof step in one subgoal may instantiate a variable which occurs in other subgoals. This means that when undoing one branch of a proof, one may obtain a subgoal that was never encountered before. This is in contrast with historical undo, where one always returns to previously attacked subgoals. The user understands local undo as an operation on the proof script, which we now explain.

### 3.3 Maintaining the proof script

There are two main reasons to maintain a proof script. First, the proof itself is *the* outcome of a session with the prover. Second, it is a convenient abstraction for the user to express control: which goal to attack, which fragment of the proof to undo because it doesn't lead to a complete proof.

To explain this, we will take the example of the HOL interface. In HOL, `THENL` is a combinator used to put together tactics. It is one of the constructors called *tacticals* ([LCF78], [Paulson87]). Its semantics is very simple. Assume that we attack a subgoal $\sigma$ with `tactic`, which now gives rise to two new subgoals $\sigma_1$ and $\sigma_2$; assume that `tactic1` proves $\sigma_1$ and `tactic2` proves $\sigma_2$. Then $\sigma$ has been proved, and the proof script is `tactic THENL [tactic1;tactic2]`. The `THENL` combinators

give a strong structure to the proof script. To represent incomplete proofs, we use the symbol $\diamond$ as a place holder, standing for a yet unspecified tactic. With that convention, we can now display the proof script before each proof step. For example, consider the consecutive application of `tactic`, then `tactic2` on the second subgoal and, finally, `tactic1` on the first subgoal. The corresponding states of the proof script are $\diamond$ then `tactic THENL [`$\diamond$`;`$\diamond$`]` then `tactic THENL [`$\diamond$`;tactic2]` and, finally, `tactic THENL [tactic1;tactic2]`.

With this notion of an incomplete script, building the proof is merely editing, in a controlled way, the proof script. This idea is probably already present in [NuPrl]. Choosing a subgoal to attack is selecting an occurrence of the symbol $\diamond$. Applying a tactic to a subgoal is substituting the corresponding $\diamond$ with this tactic and then asking the theorem prover to check the consistency of this change. If the tactic generates new subgoals, it is inserted in a `THENL` tactical, together with the appropriate number of place holders. Local undo is expressed by requesting a (possibly incomplete) tactic to be replaced by the place holder $\diamond$. Thus building a proof is a form of semantically driven structured editing.

Note that the proof script does not contain any chronological information. If desired, it is easy to add to the interface a conventional script that records the sequence of proof steps as they are sent to the proof engine, together with a path that locates them in the proof script. Historical undo is then implemented in terms of local undo.

### Improving the script

Once a proof is complete, the user has in his possession a script that proves the initial goal. This script, however, may not be in a form that the user wants to store in a library: improvements may lead to a more acceptable final script. Obviously these improvements depend on the prover. To give an example of the kinds of improvement we contemplate, consider the example of Section 1. There, we obtained the following final proof script:

```
INDUCT_TAC THENL
   [REWRITE_TAC[definition 'arithmetic' 'ADD'];
    REWRITE_TAC[definition 'arithmetic' 'ADD'] THEN
        ASM_REWRITE_TAC[]]
```

The first improvement concerns the second subproof. Instead of rewriting first with respect to the definition of ADD, then with respect to assumptions, we can do both simultaneously:

```
INDUCT_TAC THENL
   [REWRITE_TAC[definition 'arithmetic' 'ADD'];
    ASM_REWRITE_TAC[definition 'arithmetic' 'ADD'] ]
```

But in the first subproof, we could have also included assumptions in the rewriting tactic:

```
INDUCT_TAC THENL
   [ASM_REWRITE_TAC[definition 'arithmetic' 'ADD'];
    ASM_REWRITE_TAC[definition 'arithmetic' 'ADD'] ]
```

Now both cases of the inductive proof are treated by the same tactic. This is exactly what the `THEN` tactical is for. Namely the tactic `tactic1 THEN tactic2` consists in attacking all subgoals created by `tactic1` with the same tactic, `tactic2`. So we can rewrite the script as:

INDUCT_TAC **THEN**
ASM_REWRITE_TAC[definition 'arithmetic' 'ADD']

The proof is now transparent and in a form that we are willing to record. What this example shows is that, after we have obtained a proof interactively, we need computer assistance to clean it up before storing it permanently. We have not yet implemented such a facility. What is needed to support this activity is a program transformation system, that will be applied to the proof script. This means that a tool distinct from the proof engine itself should be accessible from the user interface. As mentioned in Section 2.3, this is simple with our architecture.

### 3.4 Synchronizing parallel proofs

The user can start to prove a new theorem at any time, but should not be allowed to use a theorem until it has been proved. To make things simple, we use the library of theorems as a synchronization mechanism. Only theorems recorded in the library can be used; only duly proved theorems can be entered in the library. This is perfectly safe, but not very elegant. Theorems are global in a theory, there is no notion of a *lemma* that is local to a proof. In the future, this is likely to interfere with automatic proof improvement.

## 4 Constructing a new proof step

### 4.1 Introduction

With a conventional interface, a user types in a new proof step as a command, in the proof engine's command language. While this must, of course, remain possible, we show now that the proof environment can provide significant guidance in constructing the next proof step.

### 4.2 Guided Editing

Programming environments (Centaur[Centaur92], PSG[Snelting91], Cornell Synthesizer[CSG]) have explored original ways to construct programs while respecting their syntax and type structure. Constructing a proof step is quite similar to building a program fragment. The key idea in these systems ("guided editing") is to provide upon request a menu of possibilities for any subexpression. This menu lists a finite number of legitimate choices. How these choices are computed differs from system to system. In Centaur for example, there is a generic menu facility, which can be tailored to any particular application by adding new items, such as rewrite rules, to menus.

In the case of a theorem prover, constructing a command involves roughly four kinds of objects: tactics, terms, occurrences and theorems.

#### Tactics

To explain to what extent the interface may help the user in finding a proof tactic, consider the notion of proof step in the system developed by Felty[Felty89] in a very simple and elegant prover for intuitionistic first order logic. There are fourteen basic tactics, that are associated with rules of a sequent calculus[Szabo69]. A typical rule is for example:

$$andI\_tac \quad \frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

The reading of this rule as a tactic is very simple: to prove a goal of the form $A \wedge B$ under assumptions $\Gamma$, the tactic *andI_tac* produces two subgoals, one where $A$ must be proved under assumptions $\Gamma$, and another one where $B$ must be proved under the same assumptions. Here $A$, $B$ stand for formulae, and $\Gamma$ stands for a list of formulae.

To help the user in finding a relevant tactic, the first idea is to display as a menu the list of all tactics, together with convenient means to know what each of them is for. This solution is clearly unacceptable for provers that contain a large number of tactics. To limit this list, one must use contextual information. The main source of information is the goal on which we try to apply the tactic. In Felty's prover, if the goal is, say

$$pa \vee pb \vdash \exists x\, px$$

instead of showing a list of 14 tactics, we automatically restrict the list to only two, *existsI_tac* and *orE_tac*. Furthermore, using the mouse to select a part of the goal, we can indicate whether we want to work on assumptions or the formula to the right of $\vdash$. This very coarse indication is sufficient to determine unambiguously which tactic we want to use. Ritchie aptly calls this style of interaction proof-by-pointing ([Ritchie88]).

In a large scale system, such as HOL, the situation is more complex. For a number of tactics, the method described above can be used. But there are certain tactics whose applicability cannot be characterised by pattern matching in the goal. For example, one such tactic consists in reducing arithmetic expressions to normal form whenever possible (e.g., 1+2 reduces to 3). To compute the menu of tactics available at a certain location in a goal, we may have to *guard* each tactic in the menu with an arbitrary function which, given a goal and a specific location in it, decides whether the tactic is applicable.

### Terms

Certain tactics need term arguments, i.e., mathematical formulae or expressions in a programming language. Term arguments may of course be entered as text by the user. But it is also convenient to pick such terms with the mouse in a familiar *copy and paste* manner. The architecture of the interface must clearly allow for this. The only difficulty is that the subterm is possibly extracted from a context that allows types to be inferred for its variables, and then imported into a context where there is insufficient information to do so. This happens, for example, in the HOL interface. The solution, discussed in section 2.1, is to keep a type annotation for each variable. This annotation can be made optionally visible in the user interface, but it is systematically added to the input to the proof engine. This means for example that the term $m + 0 = m$ may also be displayed as $(m : num) + 0 = (m : num)$. If $m$ is picked from this formula and used as the argument of a tactic, for example to eliminate an existential quantifier, the proof engine will not receive $m$ but the explicitly typed $m : num$.

### Occurrences

When the user points at a subexpression in a window of the interface, a generic mechanism —derived from the PPML specification— converts the physical location of the mouse to an abstract location called a *selection*. To be precise, a selection is a path in the tree that is being displayed. When the selection is performed in the goal window, the intent of the user is often to tell *where* the next command should be applied. What this means is that certain tactics take as an argument a

location in the goal, which we call an *occurrence*, rather than a subexpression. It is easy to convert selections (interface notions) to occurrences (proof engine notions).

In most proof engines, there are relatively few tactics that take occurrences as arguments, and for a good reason: when there is no serious user-interface, it is painful to compute an occurrence by hand. A system like Nuprl[NuPrl] provides some help and in our interface, occurrences are always computed automatically. To compensate for the absence of such tactics, provers provide a form of addressing by *contents*. But since a term may occur several times in a subgoal, this method is neither precise nor robust, in addition to being inefficient and unnatural. On the other hand, the technique of occurrences encounters two difficulties:

- if the proof engine doesn't have an occurrence package, it must be developed; this is not difficult. Additionally, several proof tactics must be repackaged to take occurrences rather than terms as parameters. This is a good principle, quite independently of any user-interface concerns.

- assumptions are classically thought of as included in an unordered set, rather than a list. Referring to an assumption by its rank in an ordered list seems unsatisfactory and liable to make proofs less robust.

The Coq[Coq] system permits one to give a symbolic name to each assumption, and this name may be used in occurrences. Names can also be generated automatically in Coq, but it is unreasonable to use such names in occurrences, as there is no guarantee of reproducibility. In HOL, a sophisticated scheme has been implemented. When the user refers to an assumption, the constant *context*, which is local to the proof and initially empty, is automatically updated to give a symbolic name to that assumption, if it hasn't been done already. From then on, all references to the assumption use the symbolic name. It is interesting to see that the script of the proof is extended in its declarative part as well as, more conventionally, at the place holders.

### Theorems

Only extremely simple facts get ever proved from first principles. The mathematician, the user of theorem prover organize their activity so as to maximize the reuse of theorems. Hence, one builds hierarchies of theories, starting with integers, lists, functions, groups, etc., which are of a mathematical nature, but also more applied theories to help in proving properties of programming languages or circuits. The success of a particular theorem prover may depend more on the availability of a large number of well organized theories than on any other factor. The parallel with software development is striking. A modular design, maximizing reuse, is a constant objective. In the example of section 1.2, the proof was built in theory **proof1**, on top of the hierarchy shown in Figure 1.

The graph representing ancestor theories is an interactor that may be used to browse theories. If we select the theory **arithmetic** in the graph, a window opens that displays the definitions (or the theorems) of arithmetic as a menu:

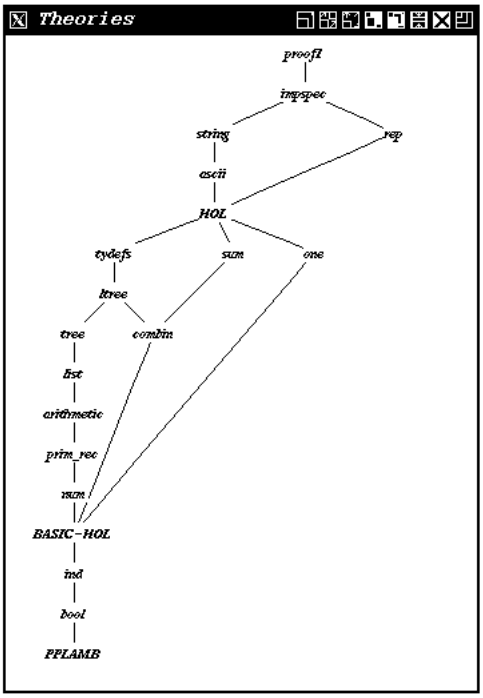
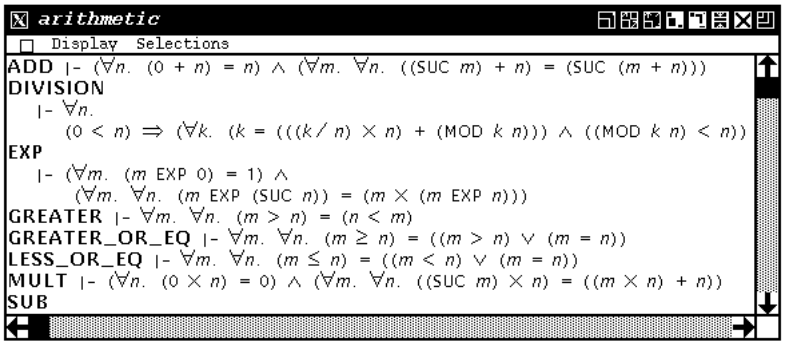



Figure 1: The theory environment for the HOL example




Selecting in this window is a way to provide arguments of type "theorem" when constructing a command. The text of a theorem may be large, so it is essential to have an elision facility in the menu. But there may also exist several dozen theorems in a theory. To use the menu facility, it is necessary to be familiar with the nomenclature used in a particular theory, or in a collection of theories. This again is a familiar software engineering concern.

Another strategy is to provide automatic *retrieval* facilities. We don't claim to understand how to do this in all generality, but we have implemented one extremely useful special case, already apparent in the example of section 1.2. When a theorem is of the form:

$$\forall x_1 x_2 \ldots x_n \; A = B$$

it may be used as a *rewrite rule*. Such theorems are used very often, to perform symbolic calculations. Their retrieval is entirely automated: the user selects a term anywhere in the goal, whether in assumptions or in the conclusion of the goal. The *retrieval* window displays what this term could

be rewritten to within a given context of theories. This context may be changed by changing the selection in the theory graph.

Retrieval must be extremely fast. Basically, the user should have the feeling that a menu of possible rewritings pops up as soon as a subterm is selected. In HOL, the mechanism of term-nets, originally implemented by Paulson, builds an automaton corresponding to the rewrite rules in a theory. It is extremely efficient. However, if the formula to rewrite is very large, the proof engine will probably send very large replies that will slow down the user-interface.

Note that the only use in displaying the name of the theorem in the retrieval window is to teach the nomenclature. The name of the theorem, and of the theory in which it was found, will be pasted automatically in the proof script, when the user selects the preferred rewrite result. Note also that two different theorems may rewrite to the same term. If we don't specify which theorem was used, the menu can be shorter and contain only the possible rewritings.

Proving theorems always involves a substantial amount of algebraic simplification, and it seems naive not to use the best tools in this area: decision procedures, simplification algorithms in certain theories, etc. In a sense, this is a problem for the proof engine. But, in terms of the user-interface, it seems also to indicate that the user-interface must be ready to organise the dialog of the user with several distinct engines, as advocated by [Kaj92]. This has consequences both in terms of protocols and in terms of man-machine interface ideas that are beyond the scope of this paper.

### 4.3 Miscellaneous ideas for the Man-machine Interface

Our experiments in constructing interfaces for different proof engines point to a number of issues in man-machine interface design. First, there is a clear need for organizing the different windows in the environment. Some windows, such as the script and the subgoal windows, are always present simultaneously because they correspond to a state of the proof. This means that, to surround our application windows, we need a powerful toolkit that includes buttons, menubars, scrollbars, dialog boxes but also multipaned windows.

The architecture allows one to display two different views of the same object or of objects that are related. Then, a selection in one window may result in a selection in the related part of another window. One way to make such relations apparent to the user is to use the background colour of the selection. Colour and fonts, used with restraint, serve also to give immediate information about the type of subexpressions. For example, theorems always show up in the same colour, no matter which window they appear in. It is our strong belief that colour, in fact, is of very significant help in making relationships between windows apparent to the user.

Another consideration comes from reflecting on the way we construct a proof step. In essence, we construct a function call by sending arguments from different views to the proof script window. The same expression, selected with a single mouse click, may be considered as a term, or as an occurrence, or even as a theorem (given a term $A$, the formula $A \vdash A$ is a theorem). In most cases, it is possible for the interface to resolve from the context which type of selection was meant, thereby relieving the user from giving this information explicitly. When the command is completely filled in, it may be sent automatically to the proof engine, with no need to return to the script window for confirmation. Taken together, these two ideas make the interface much more comfortable for an experienced user, because there is much less criss-crossing the screen with the mouse.

# 5 Conclusion and open problems

Our major conclusion is that the architecture described in this paper is *feasible* in terms of efficiency. The inconvenience of duplicating data inherent in a distributed architecture is largely compensated by judicious use of incrementality in the communication of data. In our view, *incrementality is the technique that makes a distributed architecture feasible.* An instance of that principle is the use of an incremental display in the user-interface. In [Graham92] there are proofs where subgoals have over one hundred assumptions and a two-page long conclusion. To perform such proofs in a natural fashion, incremental display of subgoals is *essential.*

The second conclusion is that there are substantial insights to gain from comparing programming environments and proof construction environments. In one direction, almost all the tools that we have included in the Centaur programming environment are useful in a prover interface. In the other direction, the way we construct proofs suggests that building programs could be surrounded with much more semantic assistance than has been considered so far.

In terms of building real environments for proofs, a most surprising result is that an interface not only eases the dialogue between the user and the prover, but that it also affects seriously our way of doing proofs. The example of the use of occurrences is typical. With an adequate interface this intuitive notion can be fully exploited to produce proofs that are more natural.

Many interesting directions in developing real proof environments must still be explored. An example is the post-processing of proofs. In Felty's theorem prover [Felty89], the proof is performed in a sequent style. When it is finished, the sequent proof is translated into a natural deduction proof [Felty91]. The natural deduction tree is then displayed in mock natural language style. In fact, as can be seen on Figure 2, full-fledged proof trees have more meta-mathematical than pragmatic interest because they do not share enough subformulae. The pseudo-natural language text, even though it is still too verbose, shares assumptions in a better way. When the proof is larger, the text grows vertically rather than horizontally and this is more familiar, at least in western culture.

Post-processing proofs, to obtain clearer and more economical proofs, is a form of reverse engineering that deserves more research. More work is also required before one can produce a reasonably terse textual form of proofs.

### Acknowledgments

Several people must be thanked for making this work possible: A. Felty made the critical modifications to her theorem proving system that were needed for an effective interface; S. Kalvala modified the goal package of HOL to allow for attacking several goals in parallel; the retrieval facility was implemented thanks to R. Boulton's experience on retrieving theorems with HOL.

Additionally, the Centaur group at INRIA Sophia–Antipolis provided generous assistance with all aspects of the system.

## References


[Centaur92] "The Centaur 1.2 Manual", I. Jacobs, *ed.*, available from INRIA – Sophia Antipolis, March 1992.

[CI89] D. Clément, J. Incerpi, "Specifying the Behavior of Graphical Objects Using Esterel", Proceedings of TAPSOFT'89, Barcelona, Spain, Springer-Verlag, LNCS 352, March 1989.

[Clement90] D. Clément, "A Distributed Architecture for Programming Environments", Proceedings of ACM SIGSOFT'90 SDE-4, Software Engineering Notes, Vol. 15, no. 6, Irvine Ca, December 1990.

[Coq] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin-Mohring, B. Werner "The Coq Proof Assistant User's Guide", INRIA Technical Report no. 134, December 1991.

[CMP91] D. Clément, F. Montagnac, V. Prunet, "Integrated Software Components: a Paradigm for Control Integration", Proceedings of the European Symposium on Software Development Environments and CASE Technology, Königswinter, Springer-Verlag LNCS 509, June 1991.

[CSG] T. W. Reps, T. Teitelbaum, "The Synthesizer generator : a system for constructing language-based editors", Springer-Verlag, 1988.

[Felty89] A. Felty, "Specifying and Implementing Theorem Provers in a Higher-Order Logic Programming Language", PhD Thesis, University of Pennsylvania, August 1989.

[Felty91] A. Felty, "A Logic Program for Transforming Sequent Proofs to Natural Deduction Proofs", in *Proceedings of the First International Workshop on Extensions of Logic Programming*, Tübingen, Germany, Springer Verlag LNAI 475, 1991.

[HOL88] M.J.C. Gordon, "HOL: A Proof Generating System for Higher-Order Logic", in *VLSI Specification, Verification and Synthesis*, G. Birtwistle, P. A. Subrahmanyam, *eds.*, Kluwer Academic Publishers, 1988.

[Graham92] B.T. Graham, "The SECD Microprocessor, A Verification Case Study", Kluwer Academic Publishers, Boston, 1992.

[Griffin88] T. Griffin, "Notational definition and top-down refinement for interactive proof development systems", PhD Thesis, Cornell University, 1988.

[Isa90] L.C. Paulson, "Isabelle: The next 700 theorem provers", in *Logic and Computer Science*, P. Odifreddi, *ed.*, pp. 361–386, Academic Press, 1990.

[Jones91] C.B. Jones, K.D. Jones, P.A. Lindsay, R. Moore, "MURAL: A Formal Development Support System", Springer-Verlag, 1991.

[Kaj92] N. Kajler, "CAS/PI: a portable and extensible interface for Computer Algebra Systems", *ISSAC'92 Proceedings*, Berkeley, July 1992.

[LCF78] M.J. Gordon, R. Milner, C. Wadsworth, "Edinburgh LCF: a mechanized logic of computation", LNCS 78, Springer-Verlag, 1978

[Maple] B. W. Char *et al.*, "MAPLE : reference manual : 5th edition", Springer-Verlag, 1992.

[Math] S. Wolfram, "Mathematica : a system for doing mathematics by computer", Addison-Wesley, 1988.

[NuPrl] R.L. Constable *et al.*, "Implementing mathematics with the Nuprl proof development system", Prentice-Hall, 1986.

[Paulson87] L. Paulson, "Logic and computation : interactive proof with Cambridge LCF", Cambridge University Press, 1987.

[Ritchie88] B. Ritchie, "The design and implementation of an interactive proof editor", PhD Thesis, University of Edinburgh, Nov. 1988.

[Snelting91] G. Snelting, "The calculus of context relations", Acta Informatica, vol. 28, no. 5, pp. 411–446, 1991.

[Szabo69] M.E. Szabo, G. Gentzen, "The Collected papers of Gerhard Gentzen", North-Holland, 1969.